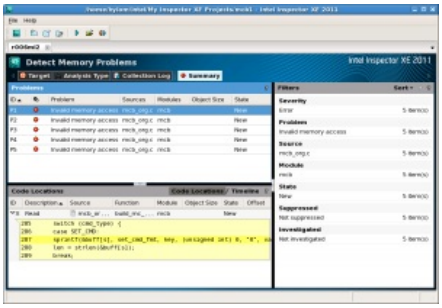
VTune Amplifier XEとInspector XEでmemcachedの高速化にチャレンジ

ソフトウェア開発において、テストやデバッグは設計やコーディング以上に重要な工程である。これらの工程において、プログラム中の問題検出やパフォーマンス解析に役立つ強力なツールがインテル Parallel Studio XEに含まれる「インテル VTune Amplifier XE」や「インテル Inspector XE」だ。本記事ではこれらのツールを用いてmemcachedのチューニングを行い、高速化を試みた事例を紹介する。

プログラム解析に役立つ「インテル VTune Amplifier XE」と「インテル Inspector XE」

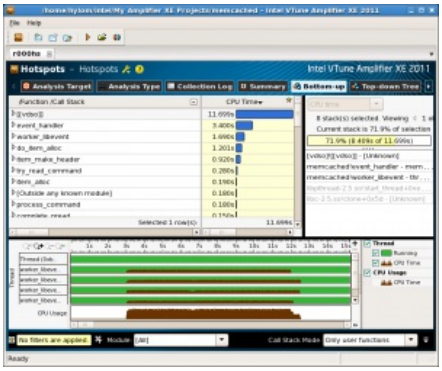
ソフトウェア開発工程においては、一般に設計→実装→テスト、という順序で開発が進められる。一般にソフトウェア開発というと、実際にプログラミングを行う実装工程が注目されがちではあるが、そのあとに行われるテストも非常に重要な工程であり、実装工程以上に時間やコストを要することも少なくない。

一般的なソフトウェアのテスト工程では、プログラム中にバグがないか、また必要となるパフォーマンスを満たしているかなどの検証や分析が行われる。これらの作業を支援する強力なツールが、インテルの最新開発ツールスイート「インテル Parallel Studio XE 2011」（以下、Parallel Studio XE）に含まれる「インテル Inspector XE 2011」（以下、Inspector XE）と「インテル VTune Amplifier XE 2011」（以下、VTune Amplifier XE）だ（**図1、2**）。



（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p1/attach/insp21.png）

図1 インテル Inspector XE 2011（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p1/attach/insp21.png）



（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p1/attach/amp07.png）

図2 インテル VTune Amplifier XE 2011（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p1/attach/amp07.png）

Inspector XEはメモリリークや不適切なメモリアクセスといったメモリ関連の問題や、デッドロックやメモリアクセスの競合といったマルチスレッドプログラムにおける問題を解析・検出するツールである。また、VTune Amplifier XEはプログラムの動きを計測し、実行に時間がかかっている個所やその原因などを表示するツールだ。

これらのツールはさまざまな分野のアプリケーション開発で役立つが、特に有用な分野の1つとして、高パフォーマンスが要

求されるサーバーソフトウェア開発が挙げられる。このようなソフトウェアではパフォーマンスを向上させるためのチューニングが必要であり、また多くのリクエストを処理するための複雑なコードは想定外の問題を生みやすい。このような状況において、Inspector XEやVTune Amplifier XEは活躍する。

以下では、実際にParallel Studio XEが有用だった事例として、「memcached」のパフォーマンス解析とチューニングを紹介する。memcachedはオープンソースのキャッシュシステムで、キーと値のペアを格納し、リクエストに応じてキーから対応する値を取り出すといった「Key-Valueストア」と呼ばれるシステムの1つだ。memcachedはWebアプリケーションの分野で非常に多く用いられており、mixiやTwitter、Flicker、Slashdotなど大手サイトでも利用されているため、その名前を聞いたことがある人も多いだろう。

memcachedは揮発性のキャッシュシステムであり、ディスクなどのストレージにはデータを保存せず、データはすべてメモリ内に格納される。そのため、実行される処理はリクエストのパースやメモリアクセスなどが大多数を占めていると予想できる。これらの処理がチューニングによって改善できるのか、また改善できる場合どのような対処を行えばパフォーマンス向上が期待できるのか、を調べることが解析の目的となる。

なお、memcachedはさまざまなOS環境で動作するが、今回はLinux環境での利用を想定して解析を行った。解析にはLinux版のParallel Studio XEを使用している。

メモリ関連の問題検出に有用なInspector XE

プログラムのチューニングを行う際にまず検討すべきなのが、プログラムのパフォーマンス測定方法だ。外部からのリクエストを受けて処理を実行するサーバープログラムの場合、なんらかの手段でサーバーに十分な負荷を与えるようなリクエストを発行する必要がある。memcachedの場合、大量のリクエストを発行してその処理時間を測定するベンチマークソフトウェアがいくつか公開されている。今回はその1つである「mcb〔http://www.interdb.jp/techinfo/mcb/〕」を用いて、パフォーマンス解析を行うこととした。

mcbは複数のスレッドを用いて連続してmemcachedにリクエストを送信し、memcachedがそれらのリクエストを処理するのにかかった時間を測定するプログラムである。ところが、筆者がこのプログラムの「version 1.0 rc2」を用いてmemcachedのベンチマークを行おうとしたところ、特定のパラメータを与えた場合のみ高確率でセグメンテーションフォールトが発生する、という問題が発生したのである。

mcbはオプションでベンチマークの条件を変更可能で、たとえば「-t」オプションで同時にアクセスを行うスレッド数を、「-n」オプションでmemcachedに対して発行するコマンド数を指定できる（**表1**）のだが、この「-n」オプションに与える値を6000以上に設定するとセグメンテーションフォールトが発生することがあり、またこの値を大きくするほど高頻度でセグメンテーションフォールトが発生する、という現象が確認できた（**リスト1**）。

表1 MCBコマンドの引数（抜粋）

オプション	説明
-c ＜コマンド＞	発行するコマンドを指定する。「set」および「add」、「get」が指定できる
-a ＜IPアドレス＞	接続するmemcachedが動いているマシンのIPアドレスを指定する
-t ＜スレッド数＞	同時に実行するスレッド数を指定する
-n ＜コマンド数＞	発行するコマンド数を指定する
-l ＜データサイズ＞	リクエストとともに送信するデータサイズの平均値を指定する

リスト1 MCBの実行例

```
$ ./memcached -m 1024m  ←memcachedを実行
$ ./mcb -c set -t 2 -n 10000 ←ベンチマークを実行：エラー終了
*** glibc detected *** ./mcb: free(): invalid next size (normal): 0x09fcc1f0 ***
======= Backtrace: =========
/lib/libc.so.6[0xbff5a5]
/lib/libc.so.6(cfree+0x59)[0xbff9e9]
./mcb[0x8049da0]
/lib/libpthread.so.0[0xd28832]
/lib/libc.so.6(clone+0x5e)[0xc67f6e]
======= Memory map: ========
00249000-0024a000 r-xp 00249000 00:00 0          [vdso]
00b72000-00b8d000 r-xp 00000000 03:03 1639306    /lib/ld-2.5.so
  :
  :
bf8c9000-bf8de000 rw-p bffe9000 00:00 0          [stack]
アボートしました
$ ./mcb -c set -t 2 -n 10000 ←同じコマンドを実行：今度は成功
condition =
        connect to 127.0.0.1 TCP port 11211
        command = set
        2 thread run
        send 10000 command a thread, total 20000 command
        data length = 1024
result =
        interval =  0.377967 [sec]
        performance =  52914.689133 [command/sec]
        thread info:
          ave. = 0.377855[sec], min = 0.377829[sec], max = 0.377881[sec]
$ ./mcb -c set -t 2 -n 10000 ←同じコマンドを実行：今度はセグメンテーション違反でエラー終了
セグメンテーション違反です
```

このように「確実には発生しないが、何回か実行するとある程度の割合で発生する」というな問題はデバッグが難しい。そこで問題を追求するため、Inspector XEを使用してデバッグを行うこととした。

インテル Inspector XEによる問題検出

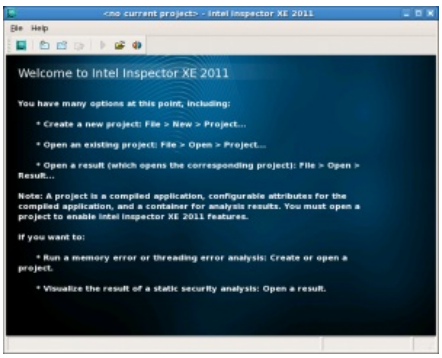
インテル Inspector XEでプログラムを解析する場合、デバッグ情報をバイナリに埋め込む「-g」オプションを付けてコンパイルを行う必要がある。mcbは単一のソースファイルのみで構成されており、今回は次のようにコンパイルを行った。

```
$ gcc -o mcb -lpthread -g mcb.c
```

コンパイルに成功したら、続けてInspector XEを起動してコマンドラインオプションなどを設定する。Inspector XEは「/opt/intel/inspector_xe_2011/bin32/」以下の「inspxe-gui」コマンドを実行することで起動できる。通常このディレクトリにはパスは通っていないので、明示的にパスを設定するか、次のようにフルパスで実行する必要がある。

```
$ /opt/intel/inspector_xe_2011/bin32/inspxe-gui
```

Inspector XEのGUIを起動すると、**図3**のようにスタートアップ画面が表示される。Inspector XEでは「プロジェクト」という単位で各種設定などを一括管理するので、まずは新たなプロジェクトを作成する。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp01.png)

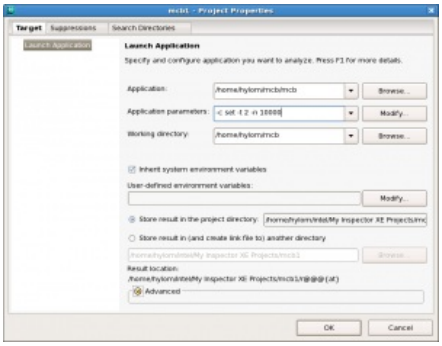
図3 インテル Inspector XE 2011のスタートアップ画面(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp01.png)

「File」メニューの「New」 - 「Project」を選択すると「Create Project」ダイアログが表示されるので、プロジェクトのディレクトリ名および作成場所を指定し、「Create Project」をクリックする（**図4**）。

(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp02.png)

図4 解析データの保存先などを指定する「Create Project」ダイアログ(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp02.png)

続けて対象となるプログラムを起動するためのコマンドラインやそのオプション、作業ディレクトリなどを設定する画面が表示されるので、これらを指定して「OK」をクリックすればプロジェクトの作成が完了する（**図5**）。

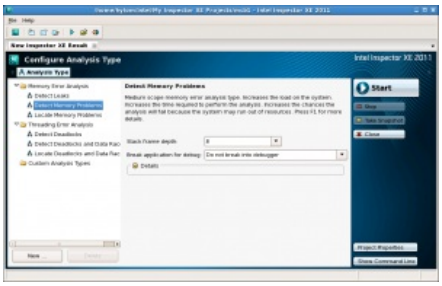


(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp04.png)

図5 プロジェクトの設定画面。実行するコマンドラインやオプションなどを設定する(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p

2/attach/insp04.png)

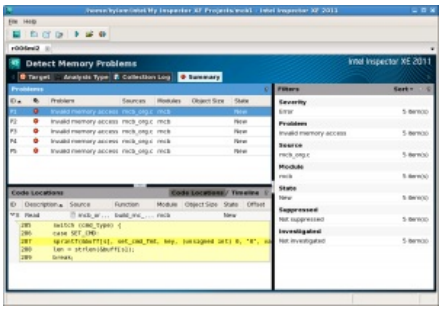
プロジェクトの作成後、ツールバーの「New Analysis Result」ボタンをクリックすると解析オプションを指定する画面が表示される（**図6**）。今回はメモリ関連のエラーを調査することが目的なので、「Memory Error Analysis」中の「Detect Memory Problems」を選択し、画面右側の「Start」ボタンをクリックすると解析が開始される。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp07.png)

図6 メモリ関連の問題を調査する「Detect Memory Problems」を選択し、「Start」をクリックする(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp07.png)

プログラムが実行され、解析が完了するとその結果がウィンドウ内に表示される（**図7**）。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp21.png)

図7 メモリ関連エラーの解析結果(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp21.png)

不正なメモリアクセスを確認する

さて、上記のようにInspector XEを実行したところ、次の**リスト2**、**3**の個所で計4つのメモリエラーが検出された。

リスト2 問題のあるメモリアクセスが検出された個所1

```
static int
build_mc_cmd(char *buff, const int buff_size, const int cmd_type,
             const int key, const char *val,
             const size_t val_len, const unsigned long int id)
{
  int len = -1;
  int s;
  memset(buff, 0, buff_size);
  s = (sysval.type == UDP) ? 8 : 0;

  switch (cmd_type) {
  case SET_CMD:
    ↓メモリ読み出しエラー発生
    sprintf(buff[s], set_cmd_fmt, key, (unsigned int) 0, "0", val_len, val);
  :
  (以下略)
  :
```

リスト3 問題のあるメモリアクセスが検出された個所2

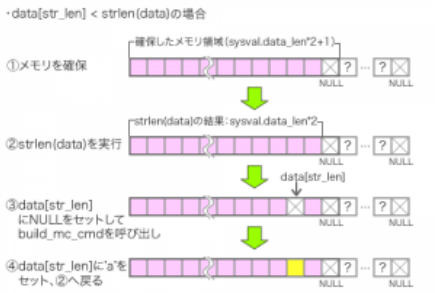
```
static void connector_thread(void *arg)
{
  const int no = (int) arg;
  int fd;
  int i, j, len, str_len;
  unsigned long int id;
  char *buff, *data;
  :
  (中略)
  :
  if ((buff = calloc(1, sysval.data_len * 2 + 100)) == NULL) { ……(1)
    elog("calloc error");
    exit(-1);
  }
  if ((data = calloc(1, sysval.data_len * 2 + 1)) == NULL) {
    elog("calloc error");
    exit(-1);
  }
  memset(data, 67, (size_t) sysval.data_len * 2); /* char'67' = 'C' */
  data[sysval.data_len * 2] = '\0';
  :
  (中略)
  :
  for (i = 0; i  sysval.command_num; i++) {
  :
  (中略)
  :
    j = 1 + (int) ((double) sysval.max_key * rand() / (RAND_MAX + 1.0));
    ↓メモリ読み出しエラー発生 ……(2)
    str_len = 1 + (int)((double)strlen(data) * rand() / (RAND_MAX + 1.0));
    data[str_len] = '\0'; ←メモリ書き込みエラー発生 ……(3)
    ↓メモリ読み出しエラー発生
    len = build_mc_cmd(buff, sizeof(buff), sysval.command, j, data, strlen(data), id);
    data[str_len] = 'a'; ←メモリ書き込みエラー発生 ……(4)

 do_cmd(fd, buff, len, id);
    if (sysval.single_command == true  sysval.type != UDP) {
      len = build_mc_cmd(buff, sizeof(buff), QUIT_CMD, 0, NULL, 0, id);
      do_cmd(fd, buff, len, id);
      do_close(fd);
    }
  :
  (中略)
  :
  }
```

検出結果を確認すると、これらの個所ではすべてポインタ「data」で指示されるメモリ領域に対してアクセスを行っていることが分かる。dataポインタはローカル変数であり、**リスト2**内のconnector_thread関数内のみで使用されている（関数の始めでメモリ領域が確保され、関数の最後で開放されている）。dataへの書き込みはconnector_thread()関数内でのみ行われて

いるため、この関数内で何か不正なメモリアクセスが行われている、ということが推測できる。

さて、以上をふまえてソースコード中の対象となる個所を確認すると、2つの問題があることに気付く。まず1つめは、build_mc_cmd関数の第2引数で与えている「sizeof(buff)」という個所である。build_mc_cmd関数の第2引数はバッファとして使用するメモリ領域のサイズを与えるものだが、「sizeof(buff)」という指定ではbuffポインタが示すメモリ領域のサイズではなく、buffポインタのサイズである「4」（32ビット＝4バイト）を返してしまう。本来はメモリ割り当て時（**リスト2**の(1)）に指定したサイズである「sysval.data_len*2+100」を指定すべきである。

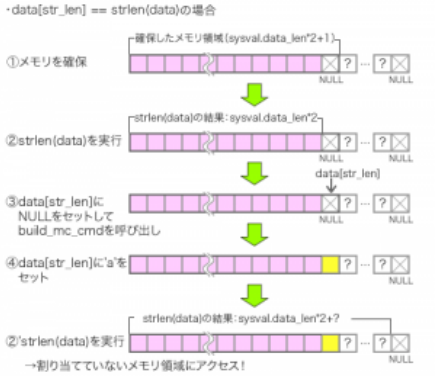
そして2つめが、(2)の個所である。ここでは「1以上strlen(data)以下」のランダムな整数値を生成してstr_len変数に格納し、続けてdatastr_len(http://sourceforge.jp/projects/hpc-parallel/wiki/str_len)にNULLをセットする、という処理を行っている（**図8**）。これはランダムな長さの文字列を生成する処理に相当する。生成した文字列はbuild_mc_cmd関数の引数として与えられた後、(4)の部分で再度NULLがあった個所に別の値を書き込むことで、バッファを復元している。ここで問題となるのが、str_lenの値がstrlen(data)と等しくなった場合である。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/mem1.png)

図8 リスト2内の(1)～(4)で行われている処理(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/mem1.png)

図9
を見ていただければ分かると思うが、「str_len == strlen(data)」となる場合、確保したメモリ領域の末尾をNULLから「a」に書き換えることになる。この部分の処理はforループ内にあり、同じ領域に対して何度も処理が繰り返されるわけだが、この場合続けて実行されるstrlenが確保しているメモリ領域外にアクセスを行い、不正な値を返してしまう。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/mem2.png)

図9 「str_len == strlen(data)」の場合、不正なメモリアクセスが発生する(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/mem2.png)

この問題はrand()で生成される乱数が特定の値の場合にのみ発生し、その確率はstrlen(data)、つまりdataバッファのサイズによって変わる。たとえばdataバッファのサイズが1024の場合、問題が発生する確率は約1000分の1程度となる。しかし、

この処理は「-n」オプションで指定した回数分だけ繰り返されるため、「-n」オプションに大きい値を指定すると問題が発生する確率が高くなるのである。

この問題の解決方法はいくつかあるが、もっとも分かりやすいのは(4)の部分を次のように修正することだろう。

```
  if (str_len != sysval.data_len * 2)
    data[str_len] = 'a';
```

このようにすれば、datastr_len(http://sourceforge.jp/projects/hpc-parallel/wiki/str_len)がメモリ領域末尾のNULLを上書きしてしまうことを回避できる。

以上の2点の修正をコードに加えた結果、mcbの実行時にセグメンテーションフォールトを発生させることはなくなり、またInspector XEで解析を行っても問題点は検出されなくなった（**図10**）。

(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp30.png)

図10 修正の結果、Inspector XEで問題は検出されなくなった(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p2/attach/insp30.png)

C/C++プログラミングにおいて、メモリ管理に関連する問題は比較的発生しやすい。たとえばセキュリティホールの原因としてよく挙げられるバッファオーバーフローは、確保していたメモリ領域を越えた位置にデータを書き込もうとして発生する。「バッファオーバーフローによる脆弱性が見つかる」というニュースが珍しくないことからも分かるとおり、熟練した開発者であっても効率良く確実にメモリの不正なアクセスを見つけることは難しい。さらに今回のように、発生の要因にランダム性があるバグは通常のデバッガでは検出や原因個所の特定が難しい。このような問題に対処するのに、Inspector XEは非常に有用なツールだといえるだろう。

アプリケーションのパフォーマンス解析に役立つVTune Amplifier XE

ベンチマークツールの問題が解決したところで、続いて本題となるmemcachedのパフォーマンス解析について紹介していこう。

プログラムのパフォーマンスチューニングを行う際に、まず確認しておきたいのが、プログラム中のどこがパフォーマンスのボトルネックとなっているか、という点だ。プログラムのどの部分で時間がかかっているかを把握しておかないと、労力をかけたのにパフォーマンスはほとんど向上しなかった、ということが発生しかねない。このようなパフォーマンスのボトルネックとなる個所（「ホットスポット」と呼ばれる）を見つけ出すためのツールが、VTune Amplifier XEだ。

memcachedのコンパイル

VTune Amplifier XEでの解析を行うには、対象となるプログラムをデバッグ情報付きでコンパイルしておく必要がある。memcachedのコンパイルは一般的なLinux/UNIX向けプログラムと同様、configureおよびmakeを使用して行う。また、memcachedはイベント処理にlibeventというライブラリを使用しており、コンパイルにはlibevent本体および関連ヘッダファイルのインストールが必要である。たとえばRed Hat Enterprise Linux（および互換ディストリビューションであるCentOS）では、次のようにして必要なファイルをインストールできる。

```
# yum install libevent-devel
```

memcachedのソースコードについては、memcached.orgよりダウンロードできる。

```
$ wget http://memcached.googlecode.com/files/memcached-1.4.5.tar.gz
```

ダウンロードしたファイルは適当なディレクトリに展開し、コンパイル用の適当なディレクトリを作成してconfigureスクリプトおよびmakeを実行する。

```
$ tar xvzf memcached-1.4.5.tar.gz
$ cd memcached-1.4.5
$ mkdir gcc1
$ cd gcc1
$ ../configure
$ make
```

以上を実行すると、configureを実行したディレクトリ内に「memcached」という実行ファイルが作成される。今回はテストのためインストール（make installの実行）は行わず、直接このバイナリを実行してmemcachedを起動することとした。

memcachedにはさまざまなオプションが用意されているが、今回のテストではシンプルに使用するメモリ容量を指定する「-m」オプションのみを指定することとした。指定する値は「1024m」（1024MB）としている。

```
$ ./memcached -m 1024m
```

なお、コンパイルに使用するコンパイラやコンパイルオプションについてはconfigureスクリプトの引数で指定することで変更できる。たとえばコンパイラにインテル C++ Composer XEを使用し、「-O3 -ipo -g」というオプションを指定する場合は次のようにする。

```
$ ../configure CC=icc LINK=xild "CFLAGS=-O3 -ipo -g" "CXXFLAGS=-O3 -ipo -g"
```

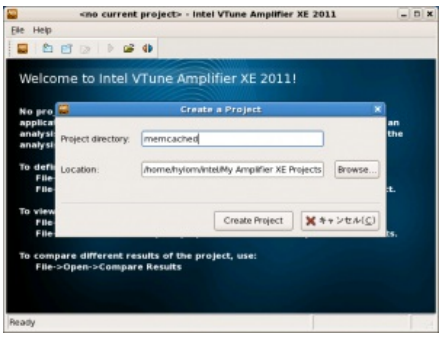
C++ Composer XEでのコンパイルやコンパイルオプションについては、インテル コンパイラーでオープンソースソフトウェアをコンパイルしよう(//magazine.sourceforge.jp/developer/article.pl?sid=09/02/25/1244257&tid=30)という記事で紹介しているので、より詳しくはそちらを参照してほしい。

プロファイリングの実行

コンパイルが完了したら、続けてVTune Amplifier XEのGUIを起動して各種設定を行う。VTune Amplifier XEのGUIは「/opt/intel/vtune_amplifier_xe_2011/bin32/」以下の「amplxe-gui」コマンドを実行することで起動できる。通常このディレクトリにはパスは通っていないので、明示的にパスを設定するか、次のようにフルパスで実行する必要がある。

```
$ /opt/intel/vtune_amplifier_xe_2011/bin32/amplxe-gu
```

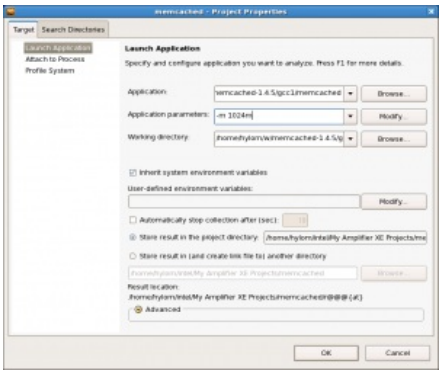
VTune Amplifier XEのプロファイリング設定は、先に説明したInspector XEでの設定手順とほぼ同じだ。まずメニューの「New」 - 「Project」で「Create Project」ダイアログを開いき、プロジェクトのディレクトリ名および作成場所を指定してプロジェクトを作成する（**図11**）。



（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp03.png）

図11 解析データの保存先などを指定する「Create Project」ダイアログ（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp03.png）

続いて対象となるプログラムを起動するためのコマンドラインやそのオプション、作業ディレクトリを指定すればプロジェクトの作成は完了だ（**図12**）。



（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp04.png）

図12 プロジェクトの設定画面。実行するコマンドラインやオプションなどを設定する（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp04.png）

プロファイリングを実行する手順もInspector XEと同様で、ツールバーの「New Analysis」ボタンをクリックし、解析オプションを指定して「Start」ボタンをクリックするだけだ。今回はホットスポットを見つけるのが目的なので、解析設定は「Algorithm Analysis」の「Hotspots」を選択した（**図13**）。

（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp06.png）

図13 ホットスポットを調査する「Hotspots」を選択し、「Start」をクリックする（http://sourceforge.

jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp06.png)

今回対象としているmemcachedはサーバープログラムであり、何らかのリクエストがないと処理は実行されない。そこで、先に紹介したmcbを使用してmemcachedにリクエストを投げることで処理を実行させることとする。リクエストはある程度負荷がかかるものである必要があるため、今回は4スレッドを使用し、平均8096バイトのデータを書き込むリクエストを10万回発生させてテストを行うこととした。mcbのコマンドラインは下記のようになる。

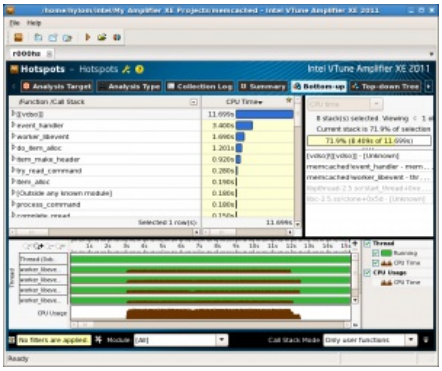
```
$ ./mcb -c set -t 4 -n 100000 -l 8096
```

なお、テストで使用したPC環境は以下の**表2**のとおりである。

表2 テストで使用した環境

要素	説明
OS	CentOS 5.5（32ビット版）
CPU	インテル Core i7 920（2.66GHz）
メモリ	3GB

mcbの実行が完了したら、「Stop」を押してプロファイリングを終了させる。するとデータの解析が行われ、その結果がウィンドウ内に表示される（**図14**）。

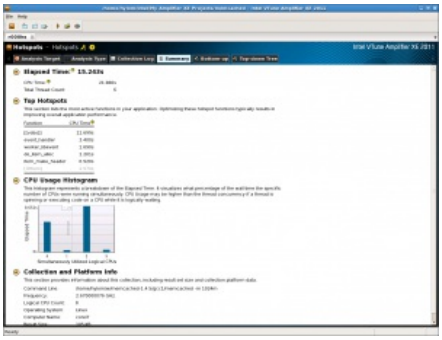


(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp07.png)

図14 プロファイル結果はグラフィカルに表示される(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p3/attach/amp07.png)

解析結果を確認する

VTune Amplifier XEでの解析結果概要は「Summary」タブ内に表示されるので、まずはここを確認する（**図15**）。ここで注目したいのが、「Top Hotspots」の部分だ。ここでは実行中に時間のかかった関数が順に表示される。

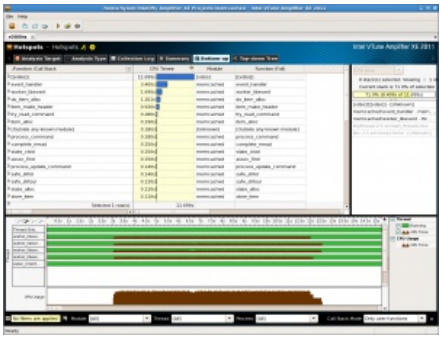


(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%A

C%E3%83%B3%E3%82%B8_p4/attach/amp09.png)

図15 「Summary」タブには解析結果の概要が表示される(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp09.png)

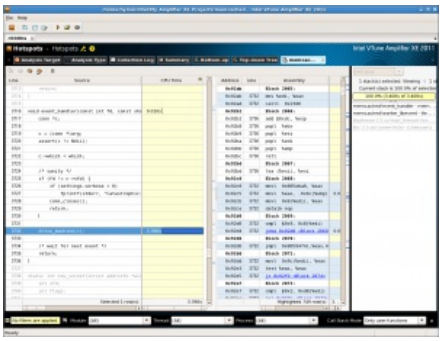
ここでもっとも時間がかかっていたのは、「[[vdso]]」の11.699秒である。この「[[vdso]]」は、「VDSO（Dynamic Shared Object」と呼ばれる仕組みを用いて実行されたLinuxカーネル命令を意味している。つまり、memcachedではLinuxカーネル命令を実行している時間がもっとも多い、ということだ。そして、次に時間がかかっている個所が「event_handler」という関数である。この関数について詳細にチェックするには、まず「Bottom-up」タブを選択して関数一覧を表示させる（**図16**）。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp08.png)

図16 「Bottom-up」タブでは、処理時間順に関数が一覧表示される(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp08.png)

続けて「event_handler」をダブルクリックすると、該当のソースコードが表示される（**図17**）。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp11.png)

図17 関数「event_handler」のソースコードとアセンブラコードが表示される(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp11.png)

さて、ここではevent_handler内の「drive_machine」関数の実行に時間がかかっていることが分かるのだが、この関数内のどこで時間がかかっているのかは表示されない。これは、この関数がコンパイラによってインライン展開されているためだ。VTune Amplifier XE内でこの関数に対応するアセンブラコードは表示されるものの、このままではより詳しい解析が難しい。そこで、ソースコードに手を加えてこの関数をインライン展開しないよう指定し、再度VTune Amplifier XEを実行することにする。

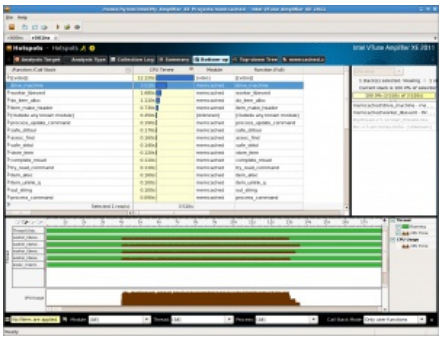
該当の関数は、memcachedのmemcached.c内で次のように宣言されている。

```
static void drive_machine(conn *c);
static int new_socket(struct addrinfo *ai);
static int try_read_command(conn *c);
```

GCCではこの宣言部分に「__attribute__((noinline))」というキーワードを付けることで、その関数をインライン展開しないように指定できる。修正後のコードは次のようになる。

```
static void drive_machine(conn *c) __attribute__((noinline));
static int new_socket(struct addrinfo *ai);
static int try_read_command(conn *c);
```

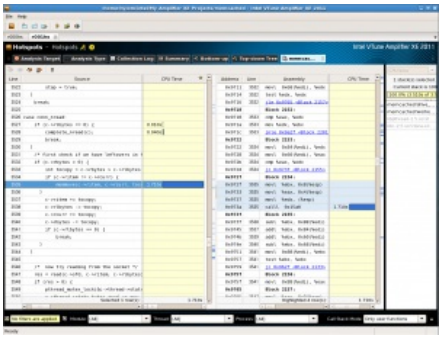
このようにソースコードを変更した後、再コンパイルを行って再度VTune Amplifier XEを実行した結果は次の**図18**のようになる。今度はインライン化を抑制したdrive_machine関数が、[[vdso]]に次ぐ時間のかかる処理となっていることが分かる。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp12.png)

図18 drive_machine関数をインライン展開しないようにした場合のVTune Amplifier実行結果(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp12.png)

さらに、drive_machine関数に注目してみると353行目の「memmove」関数呼び出しにもっとも時間がかかっていることも確認できる（**図19**）。



(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp13.png)

図19 drive_machine関数内のCPU Time表示結果(http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp13.png)

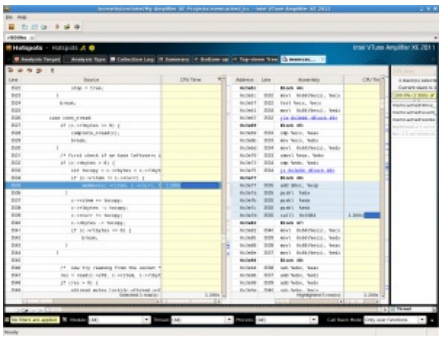
さて、以上の結果、memcachedの実行時間の大半はLinuxカーネルのAPI実行で、次にメモリ関連の処理、ということが分かった。このような場合、簡単なコード修正でパフォーマンスを向上させるのは難しい。そこで、インテル C++ Composer XEを使用してmemcachedをコンパイルし、コンパイラによる最適化でのパフォーマンス向上が見られないかを検証した。インテル C++ Composer XEには、インテル独自のライブラリを使用してC/C++標準関数を高速化するオプション「-use-intel-optimize

d-headers」が用意されている。これを利用してmemcachedをコンパイルし、VTune Amplifier XEを用いて再度パフォーマンスを測定してみた。

なお、コンパイルの際に使用したconfigureオプションは次のようになる。

```
$ ../configure CC=icc LINK=xild "CFLAGS=-O3 -ipo -g -use-intel-optimized-headers" "CXXFLAGS=-O3 -ipo -g -use-intel-optimized-headers"
```

このようにしてインテル C++ Composer XEを用いてコンパイルしたmemcachedに対し、同一の条件でVTune Amplifier XEによるプロファイリングを行った結果の「drive_machine」関数部分が次の**図20**だ。GCCでコンパイルした際には1.710秒だったmemmove関数の実行時間が、インテル C++ Composer XEによるコンパイルでは1.200秒に短縮されていることが分かる。



（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp14.png）

図20 インテル C++ Composer XEでのコンパイルでは、memmove関数の実行時間が短縮されている（http://sourceforge.jp/projects/hpc-parallel/wiki/VTune_Amplifier_XE%E3%81%A8Inspector_XE%E3%81%A7memcached%E3%81%AE%E9%AB%98%E9%80%9F%E5%8C%96%E3%81%AB%E3%83%81%E3%83%A3%E3%83%AC%E3%83%B3%E3%82%B8_p4/attach/amp14.png）

また、GCCでコンパイルしたmemcachedと、上記のコンパイルオプション付きでインテル C++ Composer XEでコンパイルしたmemcachedとでのスループットの差は次の**表3**のようになった。ここでは、約4%程度の性能向上が確認できる。

表3 GCCとインテル C++ COMPSOSER XEでコンパイルしたMEMCACHEDのスループット差

使用したコンパイラ	GCC 4.1.2	インテル C++ Composer XE 2011
コンパイルオプション	-O2 -g	-O3 -ipo -g -use-intel-optimized-headers
スループット（コマンド/秒）	54141.0	56575.5

※テストは3回おこない、その平均を最終結果としている　このようにVTune Amplifier XE 2011を利用すれば、非常に手軽にパフォーマンス解析が可能となる。また分かりやすいGUIで結果が表示され、パフォーマンス解析に慣れていないユーザーでも分かりやすく結果を確認できるのも魅力の1つと言えるだろう。

パフォーマンス改善だけでなくプログラムのデバッグや解析にも有用なインテル Parallel Studio XE 2011

以上、簡単ではあるがParallel Studio XEを用いてmemcachedのパフォーマンス解析やチューニングを試みる例を紹介した。インテルの開発ツールというと数値計算を多用する工学/科学分野やマルチメディア処理向け、という印象を持っている方もいるかもしれない。しかし今回紹介した例のように、一般的な分野においてもデバッグやパフォーマンス解析においてParallel Studio XEは非常に有用なツールとなる。

また、ここではシンプルなテストしか行っていないが、memcachedに与えるリクエストの種類やデータサイズをさまざまに変更することで、また異なる結果が出る可能性は多いにある。memcachedのチューニングに興味のある方はParallel Studio XEの無償体験版（http://www.xlsoft.com/jp/products/intel/download.html?sfwiki）を利用し、お使いの環境でコンパイルオプションや負荷のかけ方などを変えながらの実践的なチューニングにチャレンジしてみてはいかがだろうか。